Type-Directed Elaboration of Quasiquotations

A High-Level Syntax for Low-Level Reflection

David Raymond Christiansen
IT University of Copenhagen
drc@itu.dk

Abstract

Idris's reflection features allow Idris metaprograms to manipulate a representation of Idris's core language as a datatype, but these reflected terms were designed for ease of type checking and are therefore exceedingly verbose and tedious to work with. A simpler notation would make these programs both easier to read and easier to write. We describe a variation of quasiquotation that uses the language's compiler to translate high-level programs with holes into their corresponding reflected representation, both in pattern-matching and expression contexts. This provides a notation for reflected language that matches the notation used to write programs, allowing readable metaprograms.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages

General Terms Languages, Design

Keywords Quasiquotation; proof automation; metaprogramming

1. Introduction

Idris [3] is a programming language with dependent types in the tradition of Agda [16] and Cayenne [1]. An important design goal for the Idris team is to enable the construction of embedded languages that can make strong guarantees about the safety of the programs written in them, rather than requiring users of these embedded languages to write proofs themselves. If this goal is to succeed, Idris will require good tools that library authors can use to automate the construction of proofs.

One such tool is *reflection*, with which an Idris program can construct a proof object by inspecting the abstract syntax tree (AST) of a the goal type and generating an AST for the proof term. This allows developers of proof automation to write functions that might otherwise be difficult, because eliminating types through pattern matching is unsound.

Idris has a well-defined core language, called TT, and all constructions in the high-level Idris language are given their semantics by defining their translation to TT. This translation process is referred to as *elaboration*. Terms in TT are exceedingly verbose: every binder has a fully explicit type annotation, every name is fully-qualified, and there are no implicit arguments. This verbosity drastically simplifies type checking. The intention is that one need only trust this simple type checker to be able to trust the rest of the system.

Because Idris reflection works directly with TT terms, it can quickly become overwhelming. Furthermore, the correspondence between high-level Idris terms and their corresponding TT terms are not always obvious to non-expert users of the language. What appears to be a simple function application at the level of Idris code might turn out to have very complicated type-level structure or non-trivial implicit arguments. In the normal course of programming, it is good to hide this complexity and allow the user to focus on her or his programming task, rather than being overwhelmed by minutiae. However, when writing metaprograms using reflection, this becomes an unfortunate trade-off, because the core language can be difficult to connect to user-visible terms. This difficulty is especially unpleasant when using Idris's *error reflection* [5], in which Idris code can be used to rewrite the compiler's error messages before they are presented to the user. In fact, it was practical experience with error reflection that motivated the work described in this paper.

We augment the high-level Idris language with *quasiquotations*, in which the Idris elaborator is invoked to transform high-level Idris into *reflected* TT terms using the same translation that produces TT terms for the type checker. Within quasiquoted terms, *antiquotations* allow other reflected terms to be spliced into the quotation. In a pattern context, antiquotations become patterns to be matched by the reflected term at the corresponding position. These quasiquotations allow the best of both worlds: high-level syntax for the uninteresting parts, with details filled in by type-directed elaboration, along with control over the details of term construction when and if it matters.

Contribution

The contributions of this paper are:

- a novel adaptation of quasiquotations to the context of dependently typed programming with reflection that allows the use of high-level language syntax to construct and manipulate the corresponding terms in a core language;
- a description of an implementation technique for these quasiquotations in Idris, as an extension of the type-driven elaboration described in Brady's 2013 paper [3]; and
- demonstrations of the utility of these quasiquotations for proof automation and error message rewriting.

Furthermore, this paper can serve as a demonstration of how to extend a tactic-based elaborator to support a new high-level language feature. The relative straightforwardness and orthogonality of the extension can perhaps be considered an argument in favor of tactic-based elaboration as a language implementation technique.

IFL '14, October 01–03, 2014, Boston, MA, USA.

ACM 978-1-4503-3284-2/14/10. . . $15.00.
http://dx.doi.org/10.1145/2746325.2746326

2. Motivating Example

To illustrate the difference in verbosity and complexity between a term in the high-level Idris language and TT, it is sufficient to compare the natural number 1, expressed in the typical Peano encoding, in each notation. In the high-level Idris language, it is represented as `S Z`, the application of the successor operation (named `S`) to zero (named `Z`). The reflected internal structure that represents this term is:

```
App (P (DCon 1 1)
       (NS (UN "S") ["Nat", "Prelude"])
       (Bind (MN 0 "_t")
             (Pi (P (TCon 0 0)
                    (NS (UN "Nat")
                        ["Nat", "Prelude"])
                    Erased)
                 (TType (UVar -1)))
             (P (TCon 0 0)
                (NS (UN "Nat") ["Nat", "Prelude"])
                Erased)))
    (P (DCon 0 0)
       (NS (UN "Z") ["Nat", "Prelude"])
       (P (TCon 0 0)
          (NS (UN "Nat") ["Nat", "Prelude"])
          Erased))
```

The outermost constructor `App` indicates that this is an application of `S` to `Z`. The constructor `P` introduces a global name, followed by whether it is a type constructor (`TCon`), data constructor (`DCon`), or pattern-matching operator (`Ref`). Type and data constructors additionally specify their arities, along with a tag. Even this structure has been somewhat simplified. Some of the type annotations have been erased at this stage in the compiler, because they can be re-created later. These types are represented by the `Erased` constructor. `Bind` introduces a binder; in this example, we only see `Pi`, which represents dependent function spaces. Names are represented by three constructors: `UN` represents a user-provided name, `MN` represents a name generated by the machine, and `NS` indicates a name inside an explicit namespace.

Correctly constructing these reflected terms can be tedious. Additionally, one must be careful to encode precisely the right details when pattern-matching on reflected terms. In the above term, the type annotation on `S` includes a machine-generated name, because the constructor's type `Nat -> Nat` is a special case of the dependent type `(x : Nat) -> Nat`. The name `x` is constructed by the implementation during elaboration and is not predictable. In other cases, however, the particular name in a binding may be important. Other details that most pattern matches should ignore include tag values and universe level indicators. We expect that the type annotation on the `P` constructor will typically be irrelevant, though some metaprograms or proof search procedures may be able use them.

The function `isZeroR` returns `Just True` when its argument is reflection of `Z`, `Just False` when its argument is a reflection of an application of `S` to any other term, and `Nothing` when it is any other term. Such a function might be useful in a proof tactic. Even this simple function is quite verbose:

```
isZeroR : TT -> Maybe Bool
isZeroR (P _
           (NS (UN "Z") ["Nat", "Prelude"])
           _)   = Just True
isZeroR (App (P _
                (NS (UN "S") ["Nat", "Prelude"])
                _)
             n) = Just False
isZeroR _       = Nothing
```

Compare this to its equivalent for non-reflected natural numbers:

```
isZero : Nat -> Bool
isZero Z     = True
isZero (S n) = False
```

This version can return `Bool` instead of `Maybe Bool` because the type system guarantees that it will never be called with a non-`Nat` argument. However, the largest decrease in complexity comes from using Idris's high-level notation to define the patterns, rather than a datatype representing core terms.

In contrast to the somewhat verbose definition above, the quasiquotation mechanism described by this paper allows a definition of `isZeroR` that is much more like the non-reflected version:

```
isZeroR : TT -> Maybe Bool
isZeroR `(Z)    = Just True
isZeroR `(S ~n) = Just False
isZeroR _       = Nothing
```

The quotations, indicated by the backquote characters, cause the elaborator to produce a patterns that are precisely equivalent to those in the original definition of `isZeroR`. The antiquotation of `n`, indicated by preceding `n` with a tilde, causes the elaborator to treat the expression `n` normally; that is, `n` becomes an ordinary pattern variable.

3. Related Work

3.1 A Brief History of Quasiquotation

The notion of quasiquotation was invented by Quine in his 1940 book *Mathematical Logic* [12, pp. 33–37]. While ordinary quotations allow one to *mention* a phrase rather than *using* it, quasiquotations allow these quoted expressions to contain variables that stand for other expressions, just as mathematical expressions can contain variables that stand for values. In other words, a specific class of subexpression is treated as a use within a context that is mentioned. Quine used Greek letters to represent variables in quasiquotations.

The paradigmatic instance of quasiquotation in programming languages is that found in the Lisp family. Bawden's 1999 paper [2] summarizes the history and semantics of the quasiquotation mechanism found in both the Scheme family of languages and in Common Lisp. In the Lisp family, program code is represented in a uniform manner, using lists that contain either atomic data, such as symbols, strings, and numbers, or further lists. In Lisp parlance, these structures are referred to as "S-expressions". Because S-expressions are simply ordinary data, it makes sense to quote them, yielding a structure that can easily be manipulated. Additionally, most Lisps have a quasiquotation system, in which specially marked subexpressions of a quotation are evaluated, with the result substituted into the quotation. Unlike Quine's quasiquotation, the Lisp family of languages allow arbitrary expressions to be inserted into quasiquotations.

Languages outside of the Lisp family have also used quasiquotation to implement language extension. The Camlp4 system [6] provides quasiquotation for the OCaml language, among other extensions. In Camlp4, quasiquotations consist of arbitrary strings that are transformed by a *quotation expander* to either a string representing valid concrete syntax or to an abstract syntax tree. These quotations support antiquotation, which invokes the parser to read an OCaml expression or pattern inside of the quotation. Template Haskell's quasiquotations [9] work on similar principles. Both systems fully expand all quotations at compile time, and both check that the generated code is well-typed.

The MetaML family of metaprogramming facilities [15], including MetaOCaml[17] and F# [14], implement a style of quotation in which the type of quoted expressions is parameterized over

the type that would be inhabited by the the quoted expression if it were reified. These features are intended for use in staged computation. In addition to representing the types of the quoted expressions, these staging annotations feature static scope, so a quotation that contains a name contains the version of that name from the scope in which the quotation was generated.

Scala quasiquotations [13] are very much like Lisp quasiquotations. While their syntax resembles that of strings, this is a consequence of their implementation using Scala's string interpolators and they are in fact expanded to ASTs at compile time. The quasiquotations were initially intended to serve as an implementation technique for Scala macros [4], but they are also useful for both runtime code generation as well as generating program text. Scala macros closely resemble Lisp macros, in that they do not intend to allow arbitrary strings to be used as syntax, but instead implement transformations from one valid parse tree to another. Unlike Lisp, Scala programs that contain macros are type checked after macro expansion, and they are represented by a conventional AST that macros manipulate. Quasiquotations are a means of constructing and destructuring these trees using the syntax of the high-level Scala language.

Like Scala, C# is an object-oriented language with a notion of quotation [11]. In C#, quotation can be applied to an anonymous function by annotating it with the `Expression` type, which causes a datatype representing the function's AST to be generated instead of the function itself. However, this feature cannot properly be considered quasiquotation, as there is no mechanism for escaping the quotation and inserting a sub-tree that has been generated elsewhere.

3.2 Reflection, Proof Automation, and Tactic Languages

The ML language was originally developed as a metalanguage for the Edinburgh LCF proof assistant [8]. In fact, this is where the name ML is derived from. An abstract datatype was used to represent rules of inference in the underlying logic, and ML functions could then be used to construct these proofs. Higher-order functions could then be used to represent strategies for combining these functions. ML served as an expressive language for automating the construction of proofs.

Agda [16] has a notion of *reflection*, described by van der Walt and Swierstra. Agda reflection is a form a compile-time metaprogramming, where quoted terms are used to construct proof terms that are then reified and type checked at compile time. These terms are constructed through direct manipulation of the term AST, which is a simple untyped lambda calculus. Agda metaprograms can get access to reflected representations of the type that is expected at a particular source location as well as its lexical environment, and they can then use this information to construct a term matching the expected type. However, users of reflection in Agda must program with a notation matching the reflected term datatype, rather than with ordinary Agda syntax.

Coq is perhaps the best-known system that is designed to facilitate automating the construction of proofs. Early versions of Coq required that users extend the built-in collection of tactics using OCaml. LTac [7] is a domain-specific language for writing new tactics that works at a higher level of abstraction that OCaml. It provides facilities for pattern matching the syntax of arbitrary terms from Coq's term language Gallina, without these terms having been reduced to applications of constructors. Likewise, it can instantiate lower-level tactics and tacticals, which may contain Gallina terms, using portions of syntax extracted from the matched goals. Thus, LTac pattern matching can be considered a form of quasiquotation.

More recently, Ziliani et al. developed the MTac tactic language [19]. Like Agda's reflection mechanism and unlike LTac, MTac is implemented in Coq's term language, rather than being an external language. However, unlike Agda's reflection, MTac tactics use Coq's type system to classify the terms produced by tactics, and the type system can therefore catch errors in tactics. Due to the elimination restrictions and impredicativity of the `Prop` universe, one can pattern match over the structure of arbitrary terms in MTac, rather than just terms in canonical form. MTac required only minimal extensions to Coq, namely a primitive to run MTac tactics.

4. Reflection in Idris

Idris's reflection system is very similar to that of Agda. Elements of a datatype representing terms in a lambda calculus can be generated from the compiler's internal representation of TT, after which Idris programs can manipulate them or use them as input to procedures that generate new reflected terms. In addition to generating new terms, Idris allows the generation of tactic scripts through reflection, by providing a collection of base tactics as a datatype along with a primitive tactic that allows functions from an environment and a goal to a reflected tactic to be used as tactics themselves. Naturally, the tactic that applies an Idris function as a tactic is itself reflected.

Unlike Agda, the terms that are available through Idris's reflection mechanism are fully annotated with their types. Additionally, they include features of a development calculus in the style of McBride's OLEG [10], including special binding forms for holes and guesses. This representation is more complicated and more accurate than Agda's, as it maintains typing information.

5. Idris Quasiquotations

TT is a minimalist dependently-typed λ-calculus with inductive-recursive families of types and operators defined by pattern matching. The full details of TT are available in Brady's 2013 article [3].

Our quasiquotations extend the Idris^{-} language, which is a version of Idris in which purely syntactic transformations such as the translation of do-notation and idiom brackets to their underlying functions have been performed and user-defined syntax extensions have been expanded. We extend the expression language with three new productions:

$e, t ::=$ ...
| `‘(` e `)` (quasiquotation of e)
| `‘(` e `:` t `)` (quasiquotation of e with type t)
| `~` e (antiquotation of e)

The parts of a term between a quotation but not within an antiquotation are said to be *quoted*. Every antiquotation must have a corresponding quotation; that is, it is an error if the depth of nesting of antiquotations exceeds the depth of nesting of quotations. The quoted regions of a term are elaborated in the same way as any other Idris expression. However, instead of being used directly, the elaborated TT terms are first reflected. Antiquoted regions are elaborated directly into reflected terms, which are inserted as usual.

Names occurring in the quoted portion of a term do not obey the typical lexical scoping rules of names in Idris. This is because quoted terms are intended to be used in places other than where they are constructed, and their reification site may have completely different bindings for the same names. Therefore, all free names in the quoted portion are taken to refer to the global scope. Because antiquotations are ordinary terms, they obey the ordinary scoping rules of the language.

Idris supports type-driven disambiguation of overloaded names. This feature is used for everything from literal syntax for number- and list-like structures to providing consistent naming across related libraries. This is also used to allow "punning" between some types and their constructors. For instance, `()` represents both the unit type and its constructor in Idris, and `(Int, String)` can represent either a pair type or a pair of types. In ordinary Idris pro-

grams, all top-level definitions are required to have type annotations, so type information is available to aid in disambiguation. Because of this, Idris's expression language does not include type annotations on arbitrary subterms. In quasiquoted terms, however, no top-level type annotation is available. Thus, the second variant of quasiquotation above allows an explicit *goal type* to be provided. Like quoted terms, it is elaborated in the global environment. Because the goal type does not occur in the final reflected term and simply exists as a shorthand to avoid explicitly annotating names, goal types may not contain antiquotations.

6. Elaboration

The Idris elaborator, described in detail in Brady's 2013 paper [3], uses proof tactics to translate desugared Idris to the core type theory TT. A full presentation of this process is far outside the scope of this paper; however, enough details are repeated to make the elaboration of quasiquotes understandable.

6.1 The Elaboration Monad

The Idris elaborator is built on top of a library for manipulating terms in type theory. This library's primary interface is a state monad. The state consists of a current term in a version of TT that is extended with hole bindings in the style of McBride's OLEG [10] as well as metadata that is used to control the elaboration process. In particular, the elaboration state includes a hole queue, the head of which is referred to as the *focused* hole, and a collection of unsolved unification problems. The hole queue contains exactly the holes that are in the term. At the beginning of elaboration, the hole queue contains a single hole, representing the term to be constructed. As elaboration proceeds, holes are created and solved. In addition to stateful operations, the elaboration monad supports errors and error handling.

A number of meta-operations, or *tactics*, are defined in the elaboration monad. These tactics resemble the built-in proof tactics of a system like Coq. Most tactics work relative to the focused hole. In this paper, we use the following subset of Brady's [3] meta-operations:

- CHECK, which type checks a complete term;
- CLAIM, which introduces a new hole with a particular type, placing it at the rear of the hole queue;
- GET, which binds the proof state to a variable;
- FILL, which adds a guess for the focused hole, solving the imposed unification constraints;
- NEWPROOF, which obliterates the proof state and establishes a new goal;
- PUT, which replaces the proof state with a new one;
- SOLVE, which causes a guess to be substituted for its hole;
- TERM, which returns the current term; and
- UNFOCUS, which moves the focused hole to the end of the hole queue.

As a notational convention, we follow Brady [3] in letting the notation for names in the meta-language and names in the object language coincide, deferring to the reader to see which is being used. Names that occur in both contexts are metalanguage names referring to coinciding object language names. Additionally, unbound variables are taken to be fresh. When operations and their arguments occur under an arrow (e.g. $\text{CLAIM}\ \vec{a}$), it means that the operation is repeated on all the arguments in the sequence. This is similar to `mapM` in Haskell.

The meta-operations $\mathcal{E}[\![\cdot]\!]$ and $\mathcal{P}[\![\cdot]\!]$, which run relative to a proof state, elaborate expressions and patterns respectively. These operations coincide for all Idris$^-$ terms except for constants and variables. Undefined variables cause $\mathcal{E}[\![\cdot]\!]$ to fail with an error message, while undefined variables in $\mathcal{P}[\![\cdot]\!]$ are treated as pattern variable bindings. Additionally, following $\mathcal{E}[\![\cdot]\!]$, unresolved holes or variables trigger an error, while unresolved names in patterns (that is, following $\mathcal{P}[\![\cdot]\!]$) become pattern variables. Otherwise, constructors with implicit arguments (such as the length argument to the `(::)` case of `Vect`) would not be able to be pattern-matched.

Elaboration is type-directed, in the sense that the elaborator always has a goal type available and can make decisions based on this fact. However, sometimes the type will be either unknown or partially known. In these cases, unification constraints imposed by the elaboration of the term can cause the type to be solved.

In addition to the meta-operations described by Brady [3], we define four additional operations:

- ANYTHING, which introduces a hole whose type must be inferred;
- EXTRACTANTIQUOTES, which replaces antiquotations in a quasiquoted Idris$^-$ term with references to fresh names, returning the modified term and the mapping from these fresh names to their corresponding antiquotation terms;
- REFLECT, which returns a term corresponding to the reflection of its argument; and
- REFLECTP, which returns a pattern corresponding to the reflection of its argument.

The operation ANYTHING n can be defined as follows:

$$\begin{aligned}\text{ANYTHING}\ n\ =\ &\underline{\text{do}}\ \text{CLAIM}\ (n'\ :\ \texttt{Type})\\ &\quad\ \text{CLAIM}\ (n\ :\ n')\end{aligned}$$

This represents type inference because it hides the fresh name n' that is introduced for the type of n. Thus, the type must be later solved through unification with other elaborated terms. EXTRACTANTIQUOTES is a straightforward traversal of an Idris$^-$ term, replacing antiquotations with variables and accumulating a mapping from these fresh variables to the corresponding replaced subterms. The names alone are accessed by the operation $names$. REFLECT and REFLECTP each take a term and a collection of names of antiquotations (see Section 6.2) and return a quoted version of the term. Antiquotation names, however, are not quoted. Additionally, REFLECTP inserts universal patterns in certain cases — see Section 6.4

6.2 Elaborating Quasiquotations

We implement quasiquotations by extending the elaboration procedures for expressions and patterns: $\mathcal{E}[\![\cdot]\!]$ and $\mathcal{P}[\![\cdot]\!]$ respectively. Elaborating the quoted term proceeds through four steps, each of which is described in detail below:

1. Replace all antiquotations by fresh variables, keeping track of the antiquoted terms and their assigned names
2. Elaborate the resulting term in a fresh proof state, to avoid variable capture
3. Quote the elaborated term:
   (a) When not in a pattern, quote the term, leaving antiquotation variables free
   (b) When in a pattern, quote the term with additional universal patterns
4. Restore local environment and elaborate antiquotations

Replace antiquotations We replace antiquotations with fresh variables because they will need to be treated differently than the rest of the term. Additionally, the expected types of the antiquotations must be inferable from the context in which they are found, because the quotations that will fill them provide no type information. Here, variables serve their typical function: they *abstract* over the antiquoted subterms, because the term that will be constructed to fill an antiquotation at run time is unknown at elaboration time. We remember the association between the antiquoted terms and the names that they were replaced by so that the result of elaborating them can later be inserted.

Elaborate in a fresh proof state Quotations can occur in any Idris expression. However, names that are defined in quotations are resolved in the global scope, for reasons discussed in Section 5. Because the scopes of local variables are propagated using hole contexts in the proof state, it is sufficient to elaborate the quoted term in a fresh state. The replacement of antiquotations with references to fresh names means that there is no risk of elaborating the contents of the antiquotations too early. However, when the elaborator reaches these names, it will fail, because they are unknown. To fix this problem, we first use the ANYTHING meta-operation that was defined above to introduce holes for both these names and their types. Because this stage of elaboration occurs in term mode, rather than pattern mode, the elaboration will fail if the holes containing types don't get solved through unification.

Quote the term Quotation is the first step that differs between terms and patterns. In both cases, the term resulting from elaboration is quoted, with the names that were assigned to antiquotations left unquoted. However, if the term being elaborated is a pattern, then some aspects of the term are not quoted faithfully. See Section 6.4 for more information.

Elaborate the antiquotations The quoted term from the previous step is ready to be spliced into the original hole. What remains is to solve the variables introduced for antiquotations in the previous step. This is done by first introducing each name as a hole expecting a quoted term, and then elaborating them straightforwardly into their respective holes.

6.2.1 Formal Description

This four-step elaboration procedure is described in Brady's notation in Figure 1. The individual tactics that correspond to each of the steps 1–4 above are numbered. Antiquotations are replaced in the first line of the tactic script, using the previously-described operation EXTRACTANTIQUOTES (1). Then, the ordinary state monad operations GET and PUT are used to save and restore the original proof state. The region (2) bracketed by these operations corresponds to step 2 above — namely, elaboration of the quoted term in the global context, which is achieved using a fresh proof state introduced by NEWPROOF. Initially, the goal of the new proof is an unbound variable, but this variable is then bound as a hole expecting a type using the CLAIM meta-operation. The quoted term is provided with hole bindings for each of the fresh antiquotation names by the ANYTHING meta-operation. Then, the quoted term is elaborated into the main hole. If this process is successful, it will result in the hole T being filled out with a concrete type as well. The result of elaboration is saved in the variable qt, and then type checked one final time with CHECK to ensure that no errors occurred.

After the original proof state is restored with PUT, the actual quoting must be performed and the antiquotations must be spliced into the result (3). Each antiquotation name is now established as a hole of type `Term`, the datatype representing reflected terms, because the elaborated form must be a quotation. Now that the holes for the antiquotations are established, it is possible to insert the reflected term into the initial hole. The operation REFLECT is invoked, which quotes the term, leaving references to the antiquotation variables intact as references to the just-introduced holes. This quoted term is then filled in as a guess, and SOLVE is used to dispatch the proof obligation.

$$
\begin{aligned}
\mathcal{E}[\![\text{`(}\ e\ \text{)}]\!] \;=\; & \underline{\text{do}}\ (e', \vec{a}) \leftarrow \textsc{ExtractAntiquotes}\ e && (1)\\
& st \leftarrow \textsc{Get} && (2)\\
& \textsc{NewProof}\ T\\
& \textsc{Claim}\ (T\ :\ \texttt{Type})\\
& \textsc{Anything}\ (\mathit{names}\ \vec{a})\\
& \mathcal{E}[\![e']\!]\\
& qt \leftarrow \textsc{Term}\\
& \textsc{Check}\ qt\\
& \textsc{Put}\ st\\
& \textsc{Claim}\ (\mathit{names}\ \vec{a}\ :\ \texttt{Term}) && (3a)\\
& r \leftarrow \textsc{Reflect}\ qt\ \vec{a}\\
& \textsc{Fill}\ r\\
& \textsc{Solve}\\
& \textsc{ElabAntiquote}\ \vec{a} && (4)
\end{aligned}
$$

Figure 1. Elaboration of quasiquotations

Finally, the antiquotations can be elaborated (4). This is done by focusing on their holes and elaborating the corresponding term into that hole. In the above script, this is represented by the tactic ELABANTIQUOTE, which can be defined as follows:

$$
\begin{aligned}
\textsc{ElabAntiquote}\ (n, t) \;=\; & \underline{\text{do}}\ \textsc{Focus}\ n\\
& \mathcal{E}[\![t]\!]
\end{aligned}
$$

A specific elaboration procedure for antiquotations is not necessary, because programs with antiquotations outside of quasiquotations are rejected prior to elaboration.

6.3 Elaborating Goal Types

Elaborating a quasiquotation with an explicit goal type is a straightforward extension of the procedure in the previous section. After introducing a hole for the type of the term that will be elaborated prior to the actual quotation, the goal type is elaborated into this hole. Because this is occurring immediately after the establishment of a fresh proof state, names in the goal type will be resolved in the global scope, as intended.

The formal procedure is largely identical to that shown in Figure 1, with only the small addition shown in Figure 2. Thus, the lines immediately before and immediately after are included to show where the additions have occurred. This seemingly-simple change has far-reaching effects, because type information is now available to the subsequent elaboration of e'. This type information can, for instance, enable implicit arguments to be solved due to unification constraints induced by the elaboration of t.

6.4 Elaborating Quasiquotation Patterns

Quasiquotations can also be used as patterns. Recall that the operation $\mathcal{P}[\![\cdot]\!]$ is a variation of $\mathcal{E}[\![\cdot]\!]$ that is used on the left-hand side of definitions in order to elaborate patterns. The primary difference is that $\mathcal{P}[\![\cdot]\!]$ does not fail when the elaborated term contains unknown variables. Instead, it inserts pattern variable bindings for these.

It is tempting, then, to simply use the pattern elaborator in the recursive elaboration clauses of the quasiquote elaboration procedures. However, this would not work. REFLECT would simply quote these new pattern variables, leading to terms that contain explicitly quoted fresh pattern variables. Pattern elaboration must instead invoke ordinary expression elaboration when generating the

$$
\begin{aligned}
\mathcal{E}[\![\texttt{`(}\ e\ :\ t\ \texttt{)}]\!] \ =\ & \underline{\text{do}} \\
& \vdots \\
& \textsc{Claim}\ (T\ :\ \texttt{Type}) \\
& \textsc{Focus}\ T \\
& \mathcal{E}[\![t]\!] \\
& \textsc{AnyThing}\ (\mathit{names}\ \vec{a}) \\
& \vdots
\end{aligned}
$$

Figure 2. Elaborating quasiquotations with goal types

term to be quoted, but then use pattern elaboration for the antiquotations.

For practical reasons, pattern elaboration must use a specialized reflection procedure REFLECTP that introduces some universal patterns in strategic places. These universal patterns serve two purposes: preventing unnecessary pattern-matching of subterms that are uniquely determined by other subterms, and preventing elaborator-chosen features such as hidden names from making patterns too specific. In Idris, it is especially important to reduce the size of the subterms being scrutinized when possible, because the coverage checker can take significant time when checking deeply nested patterns. In particular, the constructor for references to global names contains three subterms:

- whether the name is a function, constructor or type constructor;
- the fully-qualified name; and
- a full type annotation.

The first and last of these subterms are, however, uniquely determined by the second, and they exist to simplify the type checker. Thus, when pattern matching, there is no need to check them. Additionally, the elaborator will from time to time invent a fresh name or universe variable. For example, ordinary non-dependent function types are represented in TT as dependent functions types in which the bound name is not free in the type on the right hand side. In these cases, it does not actually matter which name was chosen, because the name does not appear in the term, and matching against the specific name chosen by the elaborator could mean that a the pattern `` `(Nat -> Nat) `` did not match the input term `` `(Nat -> Nat) ``. There is no solid theoretical reason for the selection of these particular heuristics. However, they do work well in practice, and users who want to control the details of pattern matching can always override these defaults with an explicit antiquotation.

Figure 3 demonstrates the formal procedure for elaboration of quasiquotation patterns. This procedure uses two variations on previously-seen meta-operations: REFLECTP, like REFLECT, is a traversal of the resulting tree structure that implements step 4 above, and ELABANTIQUOTEP is defined as follows:

$$
\begin{aligned}
\textsc{ElabAntiquoteP}\ (n, t)\ =\ & \underline{\text{do}}\ \textsc{Focus}\ n \\
& \mathcal{P}[\![t]\!]
\end{aligned}
$$

The modifications necessary to elaborate a quasiquotation pattern with a goal type are identical to the non-pattern case.

6.5 Nested Quasiquotations

While nested quasiquotations are a useful idiom in Lisp macro programming, it is unclear to what extent they are useful in the context of proof automation through reflection. It is not particularly common to build a complex proof infrastructure around the reflection datatype itself, as reflection is primarily used to escape the confines of the type theory. However, in the interest of not introducing arbitrary restrictions, the elaboration procedure described in this section can be straightforwardly extended to support nested quasiquotations.

$$
\begin{aligned}
\mathcal{P}[\![\texttt{`(}\ e\ \texttt{)}]\!] \ =\ & \underline{\text{do}}\ (e', \vec{a}) \leftarrow \textsc{ExtractAntiquotes}\ e && (1) \\
& st \leftarrow \textsc{Get} && (2) \\
& \textsc{NewProof}\ T \\
& \textsc{Claim}\ (T\ :\ \texttt{Type}) \\
& \textsc{AnyThing}\ (\mathit{names}\ \vec{a}) \\
& \mathcal{E}[\![e']\!] \\
& qt \leftarrow \textsc{Term} \\
& \textsc{Check}\ qt \\
& \textsc{Put}\ st \\
& \textsc{Claim}\ (\mathit{names}\ \vec{a}\ :\ \vec{\texttt{Term}}) && (3b) \\
& r \leftarrow \textsc{ReflectP}\ qt\ \vec{a} \\
& \textsc{Fill}\ r \\
& \textsc{Solve} \\
& \textsc{ElabAntiquoteP}\ \vec{a} && (4)
\end{aligned}
$$

Figure 3. Elaborating quasiquote patterns

Only one modification is needed: the EXTRACTANTIQUOTES operation needs to keep track of the current quotation level. Crossing a quotation increments the quotation level, and crossing an antiquotation decrements it. Only antiquotations corresponding to the outermost quotation, i.e., only antiquotations at quotation level zero, are extracted. The remainder of the elaboration procedure is unchanged.

In the real implementation, of course, quasiquote elaboration with or without goal types and in pattern mode or expression mode is handled by one code path, with conditionals expressing the four possibilities. They are presented as four separate procedures here for reasons of clarity.

7. Examples

This section demonstrates the usefulness of quasiquotations through a number of examples, showing how the high-level notation of Idris quasiquotation simplifies their expression and reduces the need for the user to comprehend all of the details of elaboration.

7.1 Custom Tactics

In Idris, a custom tactic is a function from a proof context and goal to a reflected tactic expression. Reflected tactics are represented by the `Tactic` datatype, which has constructors such as `Exact` for solving the goal with some proof term, `Refine` for applying a name to solve the goal, leaving holes for the remaining arguments, and `Skip` which does nothing, along with tactics such as `Seq` for sequential composition and `Try` to provide a fallback in case of errors. These tactics correspond to the elaborator tactics described in Section 6.

The native tactic `applyTactic` runs a custom tactic in the scope of the current proof. In other words, its argument should be an expression of type:

```
List (TTName, Binder TT) -> TT -> Tactic
```

This construction allows Idris to be its own metalanguage for purposes of proof automation.

7.1.1 Trivial Goals

When writing proofs, it may be the case that a particular goal is completely trivial. Either the goal type is one such as `()` or the equality type that has only a single constructor, or we have

```
triv : List (TTName, Binder TT) -> TT -> Tactic
triv ctxt `(() : Type) =
  Exact `(() : ())
triv ctxt `((=) {A=~A} {B=~B} ~x ~y) =
  Exact `(the ((=) {A=~A} {B=~B} ~x ~y) Refl)
triv ((n, b)::ctxt) goal =
  if binderTy b == goal
    then Exact (P Bound n Erased)
    else triv ctxt goal
triv [] _ =
  Fail [TextPart "Decidedly nontrivial!"]
```

Figure 4. A tactic for trivial goals

```
rewrite_plusSuccRightSucc : TT -> Maybe Tactic
rewrite_plusSuccRightSucc `(plus ~n (S ~m)) =
  Just (Rewrite `(plusSuccRightSucc ~n ~m))
rewrite_plusSuccRightSucc _ = Nothing

rewrite_plusZeroRightNeutral : TT -> Maybe Tactic
rewrite_plusZeroRightNeutral `(plus ~n Z) =
  Just (Rewrite `(sym (plusZeroRightNeutral ~n)))
rewrite_plusZeroRightNeutral _ = Nothing
```

Figure 5. Rewriters for addition

a premise available with precisely the type that we desire. Idris already has a built-in tactic to solve these kinds of goals, called `trivial`. However, this built-in tactic is not extensible with support for new trivial types.

Figure 4 demonstrates an implementation of a tactic for solving trivial goals that uses our newly-introduced quasiquotations. The first case checks whether the goal is the unit type. The goal annotation is necessary because of Idris's defaulting rules, which prioritize the unit constructor during disambiguation. The second case checks whether the goal is an identity type. The explicit provision of both `A` and `B` is necessary because Idris uses heterogeneous equality, and the elaborator is unable to guess what these types are. The third case provides for a traversal of the context, checking whether a proof is already available. Finally, the fourth case causes an error to be thrown if the proof was not trivial.

7.1.2 Simplifying Arithmetic Expressions

The function `plus` that implements natural number addition is defined by recursion on its first argument. This means that certain equalities that users may consider to be trivial, such as $n + Succ(m) = Succ(n + m)$, exist as lemmas in the library that must be explicitly applied. This process is entirely tedious and can be automated. However, a general-purpose search mechanism that attempted to use the entire standard library to rewrite equalities to something easily provable would very likely be too slow and fragile to use. This is an excellent use for a custom tactic.

Indeed, a family of such tactics can be defined using a simple combinator language. In this example, we define rewriters for arithmetic expressions involving addition, zero, and successor, but the approach can easily be extended to cover more equalities.

Let a *rewriter* be a function in `TT -> Maybe Tactic`. A rewriter, when passed a goal, should either return a tactic that simplifies the goal or `Nothing`. Figure 5 demonstrates two rewriters for addition. The first uses the library proof `plusSuccRightSucc`, which expresses the identity $n + Succ(m) = Succ(n + m)$. The second uses the proof `plusZeroRightNeutral`, which expresses that zero is a right-identity of addition. Quasiquotes provide a convenient notation for both pattern-matching the goal terms and constructing the proof objects to rewrite with. Without quasiquotes, the first example would be much longer, as can be seen in Figure 6.

```
rewrite_plusSuccRightSucc : TT -> Maybe Tactic
rewrite_plusSuccRightSucc
  (App
    (App
      (P Ref (NS (UN "plus") ["Nat", "Prelude"]) _)
      n)
    (App
      (P (DCon 1 _)
         (NS (UN "S") ["Nat", "Prelude"])
         _)
      m)) =
  Just (Rewrite
          (App (App (P Ref
                       (NS (UN "plusSuccRightSucc")
                           ["Nat", "Prelude"])
                       _)
                    n)
               m))
rewrite_plusSuccRightSucc _ = Nothing
```

Figure 6. A rewriter, without quasiquotes

It is important to point out that this is a particularly *easy* case to translate. The function is monomorphic, with no implicit arguments to be solved. The types in question are first-order, with no parameters or indices. In many realistic programs, especially those in which implicit arguments must be solved, the relationship between the term to be rewritten and its low-level reflected representation might be much more difficult to discern.

Returning to the rewriting library, we can define a few simple combinators:

```
(<||>) : (TT -> Maybe Tactic) ->
         (TT -> Maybe Tactic) ->
         TT -> Maybe Tactic
rewrite_eq : (TT -> Maybe Tactic) ->
             TT -> Maybe Tactic
rewrite_nat : (TT -> Maybe Tactic) ->
              TT -> Maybe Tactic
```

The `(<||>)` operator attempts to rewrite using its left-hand rewriter. If this fails, it will attempt to rewrite with its right-hand operator. The operators `rewrite_eq` and `rewrite_nat` recurse over the structure of the goal, attempting to apply rewrite rules at each step. They apply to equality types and natural number expressions, respectively.

It is possible to derive a rewriter for equalities of expressions involving natural numbers and addition as follows:

```
rewrite_eq
  (rewrite_nat
    (rewrite_plusSuccRightSucc <||>
     rewrite_plusZeroRightNeutral))
```

This rewriter can be used in a custom tactic to repeatedly rewrite until a normal form has been reached.

7.2 Error Reflection

As described in the introduction and in a previous manuscript [5], Idris's *error reflection* allows programmatic rewriting of error messages. Just as reflection represents TT terms using an Idris datatype, error reflection represents compiler error messages as an Idris datatype that contains reflected TT terms. Then, ordi-

nary Idris functions that have been registered as error handlers will be called with reflected errors. Error handlers must have the type `Err -> Maybe (List ErrorReportPart)`, where `Err` is the type of reflected errors and `ErrorReportPart` has constructors for including strings to be printed, terms to be pretty-printed, or indented nested error messages into the rewritten error messages, allowing the rewritten errors to use the features of Idris's display routines. If an error handler returns `Nothing`, then it does not rewrite the error. While this feature is intended to provide domain-specific errors for embedded domain-specific languages, it can also be used to improve confusing error messages that result from using the standard library.

In Idris, an integer literal n is desugared to the high-level Idris expression `fromInteger` n, after which the type-driven elaboration process is used to disambiguate the name `fromInteger`. Unlike Haskell, `fromInteger` need not be a member of the type class `Num`, and it can have additional implicit arguments. These implicit arguments can be arranged such that the application only type checks when the integer literal makes sense for the type in question. The standard library contains a family `Fin : Nat -> Type` such that `Fin` n contains precisely n elements. The corresponding `fromInteger` has an implicit argument that causes the integer to be converted to a `Fin` during type checking in such a way that a failure to construct a `Fin` triggers a unification error.

This arrangement ensures that only valid integer literals can be used for `Fin`. However, the resulting type errors can be difficult to understand, as they are triggered by code that the user cannot see. Additionally, they can have different causes: the integer might not be available at compile time (for instance, `fromInteger` might be applied to a lambda-bound variable), the integer might actually be too big, or the argument to `Fin` might not be statically known.

An error handler for `fromInteger` can use quasiquotations to distinguish between these different classes of errors. The error handler relies on a few helper functions. The first converts an application of the `Nat` version of `fromInteger` to an actual `Nat`, because Idris will not necessarily normalize every term that occurs in an error, and the presence of a different `fromInteger` in the error might confuse users. This can be done by extracting the `Integer` argument, converting it to a `Nat` using `Nat`'s `fromInteger`, and then quoting the resulting `Nat` manually. This manual quotation can easily be accomplished with quasiquotations:

```
quoteNat : Nat -> TT
quoteNat Z     = `(Z)
quoteNat (S n) = `(S ~(quoteNat n))
```

Recognizing terms that consist of `fromInteger` applied to a constant integer requires a mix of quasiquotation and ordinary pattern matching. The quoted part of the pattern matches the high-level structure of the term, using the goal type to disambiguate `fromInteger` and perform type class resolution. The antiquotation of the argument ensures that the matched term actually contains a constant integer, instead of some other expression:

```
getNat : TT -> TT
getNat `(fromInteger ~(TConst (BI c)) : Nat) =
  quoteNat (fromInteger c)
getNat tm = tm
```

The constructor `TConst` represents primitive constants in `TT`, while `BI` constructs a reflected constant arbitrary-precision integer from an Idris `Integer`.

The new error messages should share a common structure: a header explaining the general context for the error, and an indented body providing specific details. This commonality is expressed in a function `mkFinIntegerErr` that wraps the specific error in this general header, in Figure 7.

```
mkFinIntegerErr : TT -> TT ->
                  List ErrorReportPart ->
                  Maybe (List ErrorReportPart)
mkFinIntegerErr lit finSize subErr
  = Just [ TextPart "When using", TermPart lit
         , TextPart "as a literal for a"
         , TermPart `(Fin ~(getNat finSize))
         , SubReport subErr
         ]
```

Figure 7. Wrapper for `Fin` literal errors

```
finHandler : Err -> Maybe (List ErrorReportPart)
finHandler (CantUnify _ tm
                      `(IsJust (integerToFin
                                  ~(TConst c)
                                  ~m))
                      _ _ _)
  = mkFinIntegerErr (TConst c) m
      [ TermPart (TConst c)
      , TextPart "is not strictly less than"
      , TermPart (getNat m)
      ]
finHandler (CantUnify _ tm
                      `(IsJust (integerToFin
                                  ~(P Bound n t)
                                  ~m))
                      _ _ _)
  = mkFinIntegerErr (P Bound n t) m
      [ TextPart "Could not show that"
      , TermPart (P Bound n t)
      , TextPart "is less than"
      , TermPart (getNat m)
      , TextPart "because"
      , TermPart (P Bound n t)
      , TextPart "is a bound variable"
      , TextPart "instead of a constant"
      , TermPart `(Integer : Type)
      ]
finHandler (CantUnify _ tm
                      `(IsJust (integerToFin
                                  ~n ~m))
                      _ _ _)
  = mkFinIntegerErr n m
      [ TextPart "Could not show that"
      , TermPart n
      , TextPart "is less than"
      , TermPart (getNat m)
      ]
finHandler _ = Nothing
```

Figure 8. An error handler for `Fin` literals

The error handler itself can be seen in Figure 8. The first case of the error handler matches errors where an actual integer is available. This means that the provided integer was simply out of bounds, so it can be reported as such. In this pattern, the `CantUnify` constructor represents a unification failure during elaboration. Its second and third arguments are the terms that could not unify, where the third argument is the one that resulted from the user's code. `IsJust` is a type family that is only inhabited when its argument is of the form `Just` x, and `integerToFin` is a function that either converts an Integer to an appropriately bounded `Fin` or returns `Nothing` — it is part of the machinery for statically ensuring that the integer was legal.

The second case matches an error where the argument is a bound variable, rather than an actual integer literal. While this will never result from an integer literal in a program, it could result from a manual application of `fromInteger`.

Finally, there is a catch-all case to handle any other failure to construct a `Fin` as well as a fall-through to avoid matching other errors: In the Idris library, this error handler is then associated with the specific argument of `fromInteger` that can generate the error, to restrict its scope and prevent false positives.

This example demonstrates how quasiquotation patterns allow a fluid transition between matching against the visible syntax of high-level Idris and the specific details of its representation in TT. Matching against visible syntax makes it far easier to read and write error handlers, yet having access to the fine details of the representation allows full control over the patterns, making it possible to do things like distinguishing between locally bound variables and references to the global context.

8. Conclusion and Future Work

This paper introduced a quasiquotation feature in the Idris language. These quotations can decrease the verbosity of reflection and allow the use of the implicit argument resolution mechanisms and type-driven overloading when constructing reflected terms. Idris's type-driven elaboration mechanism [3] was not designed with quasiquotation in mind. Nevertheless, it needed only a small amount of new code in order to handle this unforeseen extension, providing evidence that the approach can scale to new features.

The present implementation of quasiquotation has one major limitation: the elaboration of some terms in the high-level Idris language results in auxiliary definitions, which are then referenced in the elaborated TT terms. This is because, in TT, all pattern matching must occur at the top level. As an example, `case` blocks and pattern-matching `let`s are elaborated into top-level functions. Presently, the quotations of these terms simply refer to names of definitions that do not exist. Potential solutions to this problem include rejecting terms with this kind of side effect or tracking the original syntax that results in auxiliary definitions, so that two quotations of the same high-level term will refer to the same auxiliary name. Neither potential solution is entirely satisfactory.

Presently, the elaboration of quasiquote patterns introduces a number of universal patterns in invisible parts of the term where the user would not be able to predict or control the contents, such as machine-generated unused names. However, the locations at which these patterns are inserted is primarily heuristic, and not motivated by deeper theoretical concerns. Thus, they may match too many terms. A proper theoretical account of pattern matching quasiquoted terms would presumably resolve this.

Acknowledgments

I would like to thank Edwin Brady for his assistance with the Idris implementation and his comments on a previous draft of this paper. Additionally, I would like to thank my Ph.D. advisor Peter Sestoft for his comments on drafts of this paper and Eugene Burmako and Denys Shabalin for correcting my misunderstandings of Scala's quasiquotes. The comments and discussion at IFL 2014 were of great value in identifying both technical extensions to and better presentations of this work, and the feedback from the anonymous reviewers was detailed and helpful. This work was funded by the Danish National Advanced Technology Foundation (*Højteknologifonden*) grant 017-2010-3.

References

[1] L. Augustsson. Cayenne — a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250, New York, NY, USA, 1998. ACM. .

[2] A. Bawden. Quasiquotation in Lisp. In O. Danvy, editor, *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 4–12, 1999.

[3] E. Brady. Idris, a general purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[4] E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13. ACM, 2013.

[5] D. R. Christiansen. Reflect on your mistakes! Lightweight domain-specific errors. Unpublished manuscript, 2014.

[6] D. de Rauglaudre. Camlp4 reference manual, 2003. URL `http://pauillac.inria.fr/camlp4/manual/`.

[7] D. Delahaye. A tactic language for the system coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Computer Science*, November 2000.

[8] M. Gordon. From LCF to HOL: a short history. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 169–186. MIT Press, 2000.

[9] G. Mainland. Why it's nice to be quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, Haskell '07, pages 73–82. ACM, 2007.

[10] C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[11] Microsoft. Expression trees (c# and visual basic), accessed August, 2014. URL `http://msdn.microsoft.com/en-us/library/bb397951.aspx`.

[12] W. v. O. Quine. *Mathematical Logic*. Harvard University Press, revised edition, 1981.

[13] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for Scala. Technical Report 185242, École polytechnique fédérale de Lausanne, 2013.

[14] D. Syme. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*, pages 43–54. ACM, 2006.

[15] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical computer science*, 248(1):211–242, 2000.

[16] The Agda Team. The Agda Wiki, accessed 2014. URL `http://wiki.portal.chalmers.se/agda/`.

[17] The MetaOCaml Team. MetaOCaml, accessed 2014. URL `http://www.cs.rice.edu/~taha/MetaOCaml/`.

[18] P. van der Walt and W. Swierstra. Engineering proof by reflection in Agda. In R. Hinze, editor, *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, pages 157–173. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41581-4. .

[19] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in Coq. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 87–100. ACM, 2013.